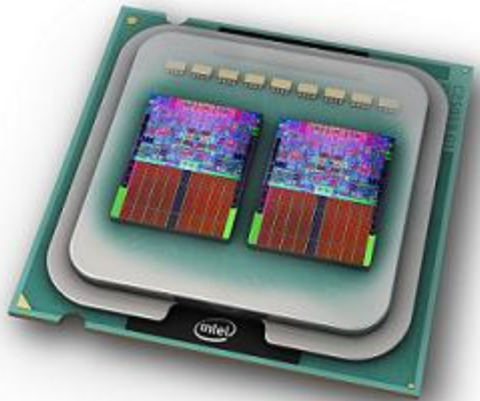
CMSC 330: Organization of Programming Languages

Multithreading

Multiprocessors

- Description
  - Multiple processing units (multiprocessor)
  - From single microprocessor to large compute clusters
  - Can perform multiple tasks in parallel simultaneously



Intel Core 2 Quad 6600

32 processor Pentium Xeon

106K processor IBM BlueGene/L

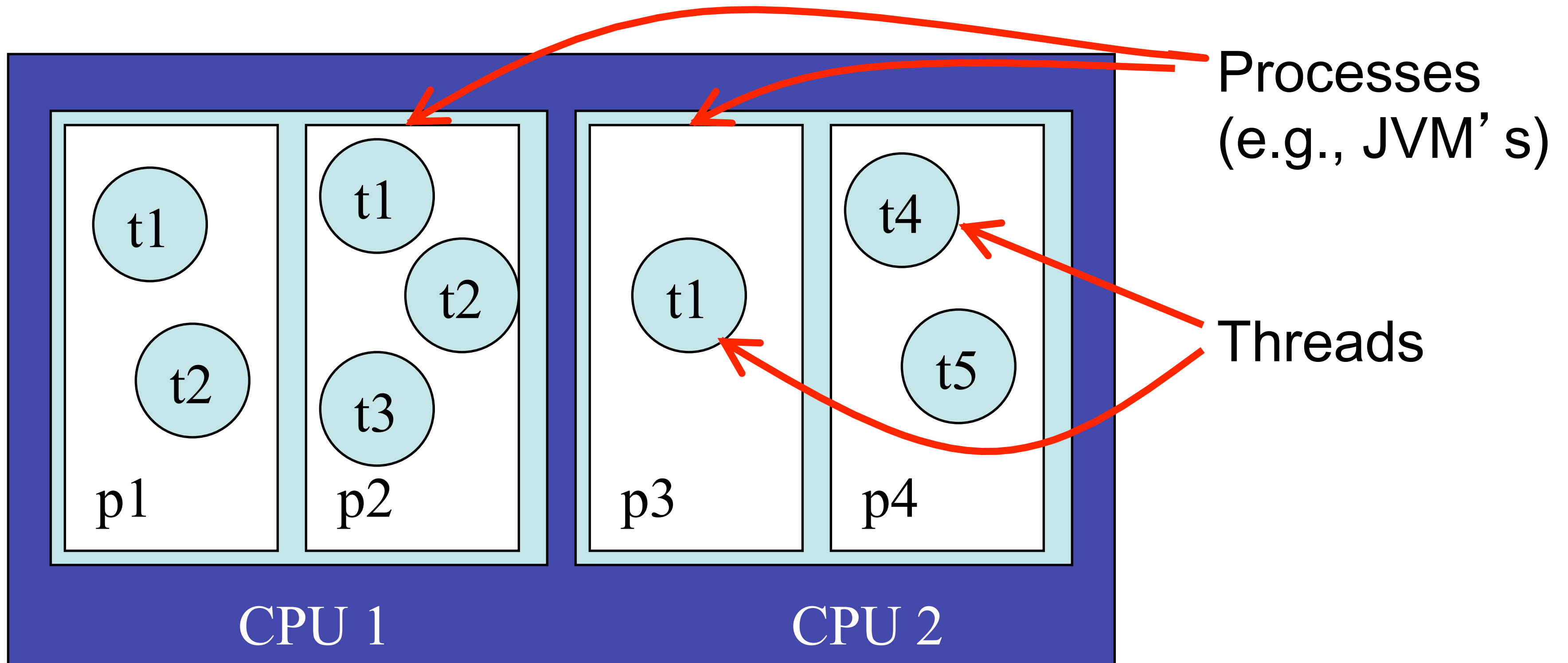
Concurrency

- Important & pervasive topic in CS
- Currently covered in
  - CMSC 132 – object-oriented programming II
    - Java threads, data races, synchronization
  - CMSC 216 – low level programming / computer systems
    - C pthreads
  - CMSC 411/430 – architectures / compilers
    - Instruction level parallelism
  - CMSC 412 – operating systems
    - Concurrent processes
  - CMSC 424 – database design
    - Concurrent transactions
  - CMSC 433 – programming language technologies
    - Advanced synchronization and parallelization
  - CMSC 451 – algorithms
    - Parallel algorithms

Parallelizable Applications of Interest

- Knowledge discovery: mine and analyze massive amounts of distributed data
  - Discovering social networks
  - Real-time, highly-accurate common operating picture, on small, power-constrained devices
- Simulations (games?)
- Data processing
  - NLP, Vision, rendering, in real-time
- Commodity applications
  - Parallel testing, compilation, typesetting, …

Computation Abstractions

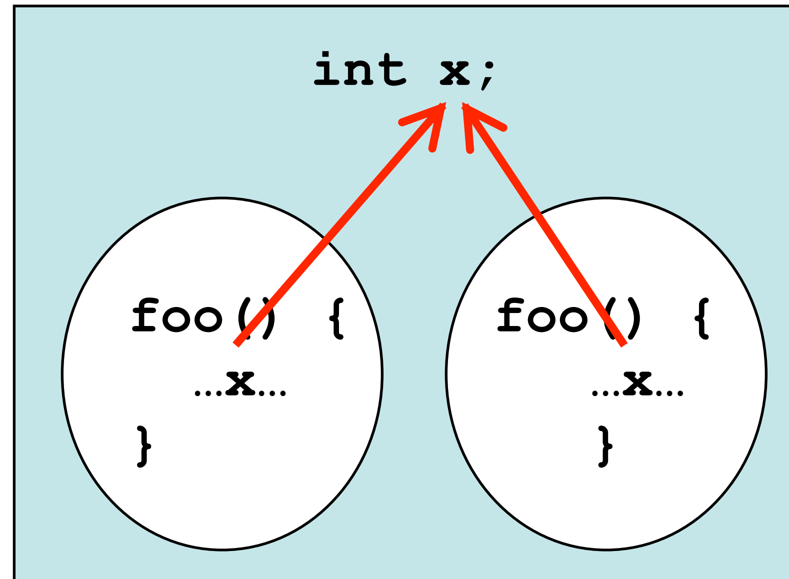


A computer

Processes vs. Threads

```
int x;
foo() {
…x…
}
```

```
int x;
foo() {
…x…
}
```



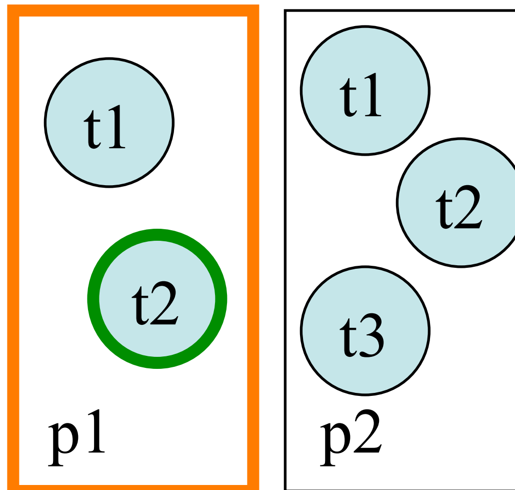
Processes do not share data

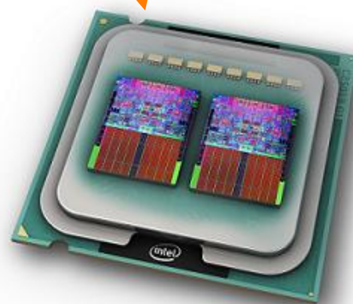
Threads share data within a process

Scheduling

OS scheduler

Thread scheduler





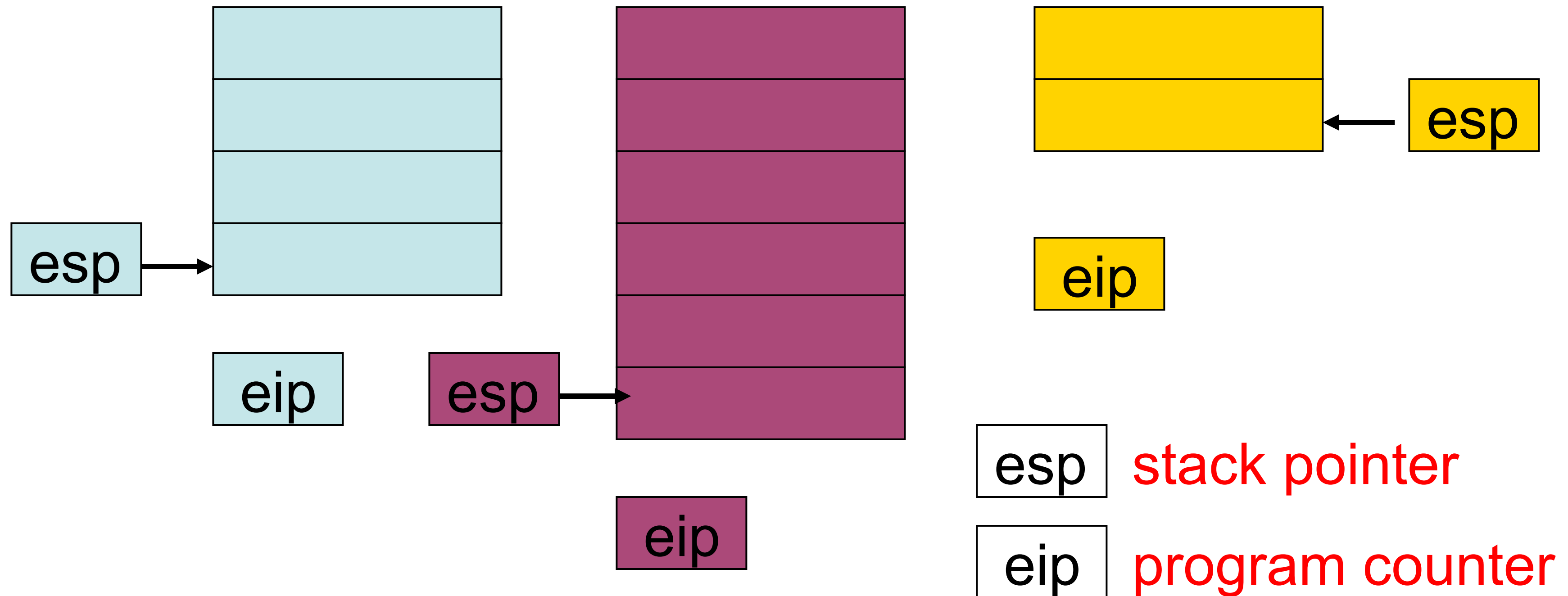
The schedulers do most of the heavy lifting.

But they don't know the semantics of what the threads are actually doing.

So, What Is a Thread?

- Fundamental unit of execution
  - All programs have at least one thread (main)
- Implementation view
  - A program counter and a stack
  - Heap and static area are shared among all threads

Implementation View

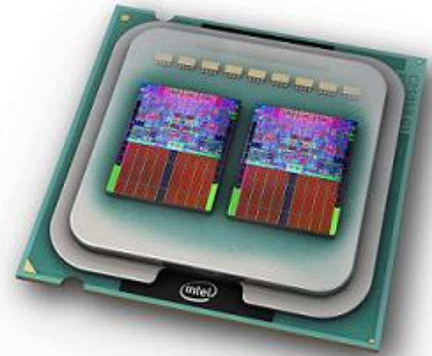


- Per-thread stack and instruction pointer
  - Saved in memory when thread suspended
  - Put in hardware esp/eip when thread resumes

Programming Processes

- Process creation is expensive
  - Stack, heap, PC, code, OS state
- Highly scalable (virtually unlimited)
- Processes may reside on separate processors
  - Sharing memory typically too expensive
- Message-passing programming paradigm
  - I/O streams, sockets, network, files
  - Cooperation key to communication (send/recv)

Programming Threads



- Thread creation is inexpensive
  - Stack, program counter, scheduler state
- Traditionally, they don't scale well (10s of threads)
  - Multicore architectures relax this limitation
- Threads reside on same physical processor
  - So memory sharing is cheap
- Shared-memory programming paradigm
  - Everything except thread local variables are shared
  - Threads communicate via shared data
  - Synchronization used to avoid data races

What Is the Big Deal about Threads?

- Conventional wisdom: threads are hard
- Main reason: non-determinism
  - Different threads execute at different speeds
  - This leads to unpredictable results
  - When you program with threads, you have to be ready for unexpected results
- The goal of the PL designer is to make this easier

Programming Languages & Threads

- Old: libraries
  - pthreads
  - Could use different libraries for different properties
- New: primitives
  - Java, Ruby, Ocaml
  - Can utilize special keywords, syntax
  - Not dependent on operating system

Which is better?

Example

- x = 0 initially. Then these threads are executed:

```
T1  y = x;          T2  z = x;
    x = y+1;            x = z+2;
```

- What is the value of x afterward   3   1   2

```
T1  y = x;     T2
    x = y+1;
                   z = x;
                   x = z+2;
```

```
T1                 T2  z = x;
                       x = z+2;
    y = x;
    x = y+1;
```

```
T1  y = x;     T2
                   z = x;
                   x = z+2;
    x = y+1;
```

```
T1                 T2  z = x;
    y = x;
    x = y+1;
                       x = z+2;
```

Data Races

- That was an example of a data race
  - Threads are "racing" to read, write x
  - The value of x depends on who "wins" (3, 1, 2)
- Languages rarely specify who wins data races
  - The outcome is nondeterministic
- So programmers restrict certain outcomes
  - Synchronization with locks, condition variables
- And they often mess up
  - Deadlocks, bugs that are hard to track down…

Programming concurrency

- Locking (lock, unlock)
  - To get mutual exclusion
- Signaling (wait, notify)
  - To not be terribly inefficient

What better way to learn than to…

Write our own programming language!

Toy Language (TL)

- Multithreaded arithmetic
  - Expressions (x+y) and commands (lock/wait/halt/if/…)
  - Language will be given an operational semantics
- Concurrency-control constructs
  - We'll capture these in operational semantics, too
  - Possibilities for adverse behavior will emerge: deadlock
- Morals
  - There's no magic
  - If you are aware of what's happening, you can write awesome concurrent code (in a real language)

TL's Expression Syntax

E ::= x | n | E *op* E $\qquad$ (*op* $\in \{+,-,/,*,$ etc.$\}$)

- x $\in$ Id is an identifier
- n is an integer
- *op* $\in \{+,-,/,*, \ldots\}$ is a set of integer operations

TL: Expression Semantics

- Results are integers ($n \in Z$)
- Remember: environments (A)
  - Map identifiers to results: $Id \rightarrow Z$
  - $A;E \Rightarrow v$ means "E evaluates to v in context of A"

These rules define '$\Rightarrow$'

–
$A;x \Rightarrow A(x)$

–
$A;n \Rightarrow n$

$E_1 \Rightarrow v_1 \qquad E_2 \Rightarrow v_2$
$E_1 \text{ op } E_2 \Rightarrow v_1 \text{ op } v_2$

TL: Sequential Commands Syntax

- TL programs have expressions and commands
  - Commands modify environment, implement control flow
  - Later they will also create threads, etc.

- Syntax of sequential (i.e. no threads) commands
  - Let x be an Id, E an expression, ℓ an Id ("label")
  - Then a command has following form

```
C :=   skip
   |   halt
   |   x = E
   |   goto ℓ
   |   if E != 0 then goto ℓ
```

  - Let Cmd be the set of all commands

TL: Sequential Programs Syntax

- Programs will be sequences of (possibly labeled) commands

```
       sum = 0
top:   sum = sum + B          } repeat...
       B = B-1                  until
       if B != 0 goto top
       halt
```

- What this example program does:
  - If B ≥ 1 when the program starts then sum = 1 + … + B

TL: Sequential Program Semantics

- We usually think of expressions as being atomic
  - A;E ⇒ v means to evaluate E "all at once"
- But atomicity isn't free
  - For non-atomic expressions, we'll need locks, etc.
  - We need a way to model "smaller steps"

Small-step semantics

- Large step: $A;E \Rightarrow v$

Need to define this

- Small step: $P \vdash \langle m, pc \rangle \rightarrow \langle m', pc' \rangle$
  - P: program
  - m, m': memory
  - pc, pc': program counter

Need to define these

- In one execution step of the program P:
  - memory changes from m to m', and
  - program counter changes from pc to pc'

TL: Sequential Program Semantics

- Semantics of programs will be operational, but small-step rather than large-step
  - Expression semantics is large-step
    - Relation $A;E \Rightarrow v$ says how to evaluate E "all at once"
    - Rules explain how to evaluate an expression in terms of what sub-expressions of E evaluate to
  - Small-step semantics explains how to evaluate in terms of small, atomic steps
    - Relation will have form $P \vdash \langle m,pc \rangle \rightarrow \langle m',pc' \rangle$
    - Idea: if "memory" is m and program counter is pc, then in one execution step memory can change to m' and program counter to pc'
- Why small-step?
  - It will turn out to be essential when considering threads

Program Counters

- A program counter is a unique ID for a command
  - The next command to execute in a program P
  - We'll also need a way to represent "P has terminated"

- Formally, the set PC of program-counter values is: $PC = N \cup \{ \bullet \}$
  - N is {0,1,…}; in essence the line number
  - The value • represents termination

Programs

- Programs map program counters to commands
  - Each PC corresponds to at most one command
  - Two programs are the same if they do the same things in the same order
- Formally:
  - P is a function $N \rightarrow (Cmd \cup (Id \times Cmd))$
  - P(pc) represents command at program counter pc
  - It may or may not have a label $\in$ Id
  - Note: P(•) is not defined!

Example Program

- Recall:

```
       sum = 0
top:   sum = sum + B
       B = B-1
       if B != 0 goto top
       halt
```

- In our definition:

N	Cmd ∪ (Id × Cmd)
0	`sum = 0`
1	⟨`top, sum = sum + B`⟩
2	`B = B-1`
3	`if B != 0 goto top`
4	`halt`

Labels

- We need a formal definition of a label (and "goto") if we want to define a program's semantics
- This can be done with a label table
  - Maps an id (like "top") to a program counter
- Formally, the label table L_P for program P is defined as:
  - $L_P(\ell)$ = the smallest number i such that $P(i) = \langle \ell, c \rangle$ (for some command c)

Id	N
`top`	1

Memories

- A mapping from ids to values (Id → Z)
- Notation: m[x:=v]
  - Same as m, but x is now mapped to v

```
        sum = 0
top:    sum = sum + B
        B = B-1
        if B != 0 goto top
        halt
```

Id	Z
B	5

Id	Z
B	5
sum	0

Id	Z
B	5
sum	5

Id	Z
B	4
sum	5

Id	Z
B	4
sum	9

Small-Step Semantics for Sequential TL

- We can now define the small-step semantics!
- Judgments will have form
  - $P \vdash \langle m, pc \rangle \rightarrow \langle m', pc' \rangle$

    "P performs one execution step, changing memory from m to m' and program counter from pc to pc'
  - Note that $pc' \in PC$, and may therefore have value •
- Definition will be given via rules, as before
  - Note that P(pc) is defined only if $pc \in N$, i.e. $pc \neq$ •

Small-Step Semantics for Sequential TL

- We can now define rules for our commands:

```
C :=   skip
   |   halt
   |   x = E
   |   goto ℓ
   |   if E != 0 then goto ℓ
```

- Judgments will have the form

$$P \vdash \langle m, pc \rangle \rightarrow \langle m', pc' \rangle$$

  - "P performs one execution step, changing memory from m to m' and program counter from pc to pc' "

Rules for Sequential TL Semantics

$$\frac{P(pc) = \texttt{skip}}{P \vdash \langle m, pc\rangle \rightarrow \langle m, pc+1\rangle}$$

$$\frac{P(pc) = \texttt{halt}}{P \vdash \langle m, pc\rangle \rightarrow \langle m, \bullet\rangle}$$

$$\frac{P(pc) = x = E \quad m;E \Rightarrow v}{P \vdash \langle m, pc\rangle \rightarrow \langle m[x:=v], pc+1\rangle}$$

$$\frac{P(pc) = \texttt{goto}\ \ell \quad L_P(\ell) = pc'}{P \vdash \langle m, pc\rangle \rightarrow \langle m, pc'\rangle}$$

$$\frac{P(pc) = \texttt{if}\ E\ \texttt{!=}\ \texttt{0}\ \texttt{goto}\ \ell \quad m;E \Rightarrow v \quad v \neq 0 \quad L_P(\ell) = pc'}{P \vdash \langle m, pc\rangle \rightarrow \langle m, pc'\rangle}$$

$$\frac{P(pc) = \texttt{if}\ E\ \texttt{!=}\ \texttt{0}\ \texttt{goto}\ \ell \quad m;E \Rightarrow 0}{P \vdash \langle m, pc\rangle \rightarrow \langle m, pc+1\rangle}$$

Rules for Sequential TL Semantics

$$\frac{P(pc) = \texttt{skip}}{P \vdash \langle m, pc \rangle \rightarrow \langle m, pc+1 \rangle}$$

Nothing to see here: just move to the next command (memory unchanged)

$$\frac{P(pc) = \texttt{goto}\ \ell \qquad L_P(\ell) = pc'}{P \vdash \langle m, pc \rangle \rightarrow \langle m, pc' \rangle}$$

goto's assign the program counter. Memory untouched.

Rules for Sequential TL Semantics

$$\frac{P(pc) = \texttt{halt}}{P \vdash \langle m, pc \rangle \rightarrow \langle m, \bullet \rangle}$$

Stop execution by clearing out the program counter. (The other way to stop is to move pc to something undefined, but this rule doesn't say that.)

$$\frac{P(pc) = x = E \qquad m;E \Rightarrow v}{P \vdash \langle m, pc \rangle \rightarrow \langle m[x:=v], pc+1 \rangle}$$

Make the assignment (add it to memory) and move to the next command

Rules for Sequential TL Semantics

$$\frac{P(pc) = \texttt{if } E \texttt{ != 0 goto } \ell \qquad m;E \Rightarrow v \qquad v \neq 0 \qquad L_P(\ell) = pc'}{P \vdash \langle m, pc \rangle \rightarrow \langle m, pc' \rangle}$$

Evaluate E; perform the goto *if* E != 0

$$\frac{P(pc) = \texttt{if } E \texttt{ != 0 goto } \ell \qquad m;E \Rightarrow 0}{P \vdash \langle m, pc \rangle \rightarrow \langle m, pc+1 \rangle}$$

Perform the goto *only if* E != 0
Otherwise, just go to the next command.

Program Execution

- Run program via → until you can't run it any farther

- Program execution is a sequence:

$$\langle m_0, pc_0 \rangle \rightarrow \langle m_1, pc_1 \rangle \rightarrow \langle m_2, pc_2 \rangle \rightarrow \ldots \rightarrow \langle m_{n-1}, pc_{n-1} \rangle \rightarrow tc$$

  - $m_0 = m$ and $pc_0 = 0$ (starting configuration)
  - For all $i < n-1$, $P \vdash \langle m_i, pc_i \rangle \rightarrow \langle m_{i+1}, pc_{i+1} \rangle$
  - May run forever

- A terminal configuration (tc) exists if
  - The program halted: $tc = \langle m', \bullet \rangle$
  - The program attempted an illegal operation: $tc = \langle m', pc' \rangle$ but $pc' \notin dom(P)$

TL: Adding Threads

- So far TL is purely imperative
  - Assignment statements, mutable memory
  - One “thread of control”
- To enrich TL with threads, we will
  - Add a command for thread creation
  - Enrich the small-step semantics to deal with multiple threads of control

Adding Threads to TL: Syntax

- New command to be added to Cmd:

`spawn` ℓ

- Intended meaning
  - Create a new thread of control, starting at label ℓ
  - The old thread and the new one now may execute concurrently

$P \vdash \langle m, pc \rangle \rightarrow \langle m', pc' \rangle$ will no longer be enough

TL: Thread Semantics Preliminaries

- $P \vdash \langle m, T \rangle \rightarrow \langle m', T' \rangle$
  - T, T' : thread pools
  - A thread pool will have multiple pc values stored in it

- Thread pool definition
  - T(i) = the pc value for thread i
  - T is a function in $\{0,\ldots,n\text{-}1\} \rightarrow N$

Thread Pool Operations

- Some new notation
  - T[i:=pc] updates the program counter of thread i to pc
  - T+[pc] adds a new thread with initial PC of pc

- Formally, let T be a thread pool of size n:
  - Update: If $0 \leq i < n$ then T[i:=pc] is the thread pool that is just like T except that i is mapped to pc.
  - Thread addition: If $pc \in N$ then T+[pc] is a thread pool of size n+1 given by

    $(T+[pc])(n) = pc$ and $(T+[pc])(i) = T(i)$ if $i < n$.

TL: Thread Semantics

$$\frac{T(i) = pc \qquad P \vdash \langle m, pc \rangle \rightarrow \langle m', pc' \rangle}{P \vdash \langle m, T \rangle \rightarrow \langle m', T[i:=pc'] \rangle}$$

If a thread i can execute in isolation,
it can execute within pool T

$$\frac{P(T(i)) = \mathbf{spawn}\ \ell \qquad L_P(\ell) = pc'}{P \vdash \langle m, T \rangle \rightarrow \langle m, (T[i:=pc+1])+[pc'] \rangle}$$

1. Creates a new thread in T;
2. Moves to the next command

How do TL threads actually run?

- Sequences of execution steps, as before
  - $\langle m_0,T_0\rangle \rightarrow \langle m_1,T_1\rangle \rightarrow \langle m_2,T_2\rangle \rightarrow \ldots \rightarrow \langle m_n, \emptyset\rangle$
  - Initially, T_0 has a single thread (main)
- Rules said nothing about which threads run when
  - In practice, schedulers do this
  - Usually OS-specific; languages don't specify it
  - BUT we did say how a *single* thread runs (pc → pc', etc.)
- So TL programs exhibit:
  - Parallelism: several execution steps can happen at once
  - Nondeterminism: threads can run in any order
- Now we need to provide a way for threads to coordinate

Locks

- Language designers limit non-determinism by introducing concurrency-control constructs
  - Make some parts deterministic = more predictable
  - Trade-off: reducing concurrency can reduce performance
- Common concurrency-control feature: locks
  - They “guard” shared resources that shouldn’t be accessed by more than one thread at a time
- The gist:
  - At most one owner at a time
  - If someone else owns it, you block
  - When you’re done with it, release ownership

TL: Adding Locks

- Adding two new commands to Cmd
  - **`acquire`** ℓ (ℓ is an id): an attempt to acquire lock ℓ
  - **`release`** ℓ releases lock ℓ
- Semantics: need to model which locks are held
  - $P \vdash \langle m, T \rangle \rightarrow \langle m', T' \rangle$ is not enough
- Locked sets
  - $L, L' \subseteq Id$ is a set of locks that are currently held
  - $P \vdash \langle m, T, L \rangle \rightarrow \langle m', T', L' \rangle$
  - **`acquire`**, **`release`** will manipulate locked sets

TL: Lock Semantics

$$\frac{P(T(i)) = \texttt{acquire}\ \ell \qquad \ell \notin L}{P \vdash \langle m, T, L \rangle \rightarrow \langle m, T[i:=pc+1], L \cup \{ \ell \} \rangle}$$

Applies if lock ℓ is available;
otherwise, the thread blocks

$$\frac{P(T(i)) = \texttt{release}\ \ell}{P \vdash \langle m, T, L \rangle \rightarrow \langle m, T[i:=pc+1], L - \{ \ell \} \rangle}$$

In our toy language, anyone can release a lock;
in practice, only a thread with a lock can release it

Program Execution with Locks

- As before, programs execute via sequences of → steps
- Initial configuration: $\langle m, T_0, L_0 \rangle$ where $L_0 = \emptyset$

```
0: acquire xlock
1: y = x
2: x = y+1
3: release xlock
```

```
0: acquire xlock
1: z = x
2: x = z+2
3: release xlock
```

L: {} → {xlock} → {} → {xlock} → {}

Signaling

- Common scenario:
  - You get the lock
  - You realize it's no good to you yet (buffer is empty)
  - Go to sleep: "wake me up when there's work to do"
- Virtually every threading model supports this
  - Condition variables
  - Wait/notify
- Conceptually simple; semantically weird
  - Let's add it to TL!

Signals in TL

- Let's add two commands:
  - **`wait`** ℓ represents a desire to go to sleep until notification about lock ℓ
  - **`notifyAll`** ℓ awakens all processes waiting notification about lock ℓ

- How **`wait`** should work
  - Acquire ℓ
  - If desired property doesn't hold, release ℓ and sleep
  - After being woken up (by **`notifyAll`**), repeat

TL: Signal Semantics Preliminaries

- $P \vdash \langle m,T,L \rangle \rightarrow \langle m', T', L' \rangle$ is not enough
  - Need to keep track of who's waiting for what
- $P \vdash \langle m,T,L,W \rangle \rightarrow \langle m',T',L',W' \rangle$
- W, W' are wait sets
  - A subset of $Id \times N \times \{a,w\}$
    - ⟨what, who, asleep/woken up⟩
  - $\langle \ell,i,a \rangle \in W$: thread i is asleep, waiting for ℓ
  - $\langle \ell,i,w \rangle \in W$: thread i was "woken up" (can re-acquire ℓ)
- Some notation
  - awake(i,W) : i is awake; i.e., it is not in W
  - $W \uparrow \ell$: W, but with all threads waiting on ℓ "woken up"
    - Change all $\langle \ell,i,a \rangle \in W$ to $\langle \ell,i,w \rangle$

Rules for wait, notifyAll

$$\frac{P(T(i)) = \texttt{wait}\ \ell \qquad \ell \in L \qquad awake(i,W)}{P \vdash \langle m, T, L, W\rangle \rightarrow \langle m, T[i{:=}pc{+}1], L - \{\ell\}, W \cup \{\langle \ell, i, a\rangle\}\rangle}$$

ℓ must be locked; it is released as part of completing wait

$$\frac{P(T(i)) = \texttt{notifyAll}\ \ell \qquad \ell \in L \qquad awake(i,W)}{P \vdash \langle m, T, L, W\rangle \rightarrow \langle m, T[i{:=}pc{+}1], L, W \uparrow \ell\rangle}$$

every thread waiting on ℓ is awakened!

$$\frac{\langle \ell, i, w\rangle \in W \qquad \ell \notin L}{P \vdash \langle m, T, L, W\rangle \rightarrow \langle m, T, L \cup \{\ell\}, W - \{\langle \ell, i, w\rangle\}\rangle}$$

if thread i in W has been "woken up" and the lock ℓ it wants is free, then i acquires the lock and is removed from the waiter set

TL: Signal Semantics

- Need to modify all our semantics so far:
  - Add W to source/target of each judgment
  - Every rule needs an extra hypothesis: awake (i,W)
    - An execution step can only happen for thread i if it is active!

$$\frac{P(T(i)) = \texttt{acquire}\ \ell \qquad \ell \notin L}{P \vdash \langle m, T, L\rangle \rightarrow \langle m, T[i{:=}pc{+}1], L \cup \{\ell\}\rangle}$$

$$\frac{P(T(i)) = \texttt{acquire}\ \ell \qquad \ell \notin L \qquad \mathrm{awake}(i,W)}{P \vdash \langle m, T, L, W\rangle \rightarrow \langle m, T[i{:=}pc{+}1], L \cup \{\ell\}, W\rangle}$$

Entire Rule Set for Threads, Locks, Signals

$$\frac{P \vdash \langle m, P(T(i)) \rangle \rightarrow \langle m', pc' \rangle \qquad awake(i,W)}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m', T[i{:=}pc'], L, W \rangle}$$

$$\frac{P(T(i)) = \mathtt{spawn}\ \ell \qquad LP(\ell) = pc' \qquad awake(i, W)}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m, (T[i{:=}pc+1]) + [pc'], L, W \rangle}$$

$$\frac{P(T(i)) = \mathtt{acquire}\ \ell \qquad \ell \notin L \qquad awake(i,W)}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m, T[i{:=}pc+1], L \cup \{\ell\}, W \rangle}$$

$$\frac{P(T(i)) = \mathtt{release}\ \ell \qquad awake(i, W)}{P \vdash \langle m, T, L \rangle \rightarrow \langle m, T[i{:=}pc+1], L - \{\ell\} \rangle}$$

$$\frac{P(T(i)) = \mathtt{wait}\ \ell \qquad \ell \in L \qquad awake(i,W)}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m, T[i{:=}pc+1], L - \{\ell\}, W \cup \{ \langle \ell, i.\ a \rangle \} \rangle}$$

$$\frac{P(T(i)) = \mathtt{notifyAll}\ \ell \qquad \ell \in L \qquad awake(i,W)}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m, T[i{:=}pc+1], L, W \uparrow \ell \rangle}$$

$$\frac{\langle \ell, i, w \rangle \in W \qquad \ell \notin L}{P \vdash \langle m, T, L, W \rangle \rightarrow \langle m, T, L \cup \{\ell\}, W - \{ \langle \ell, i, w \rangle \} }$$

Oh my, what have we done…

- We have designed a programming language with formal semantics for multithreading!
- This exposed:
  - What is really happening in these abstractions
  - What state is needed to implement them
    - memory, labels, thread pools, lock sets, wait sets
- Think about:
  - What should/shouldn't go into a language? Why?

Thread API concepts

- Thread management
  - Creating, killing, joining (waiting for) threads
  - Sleeping, yielding, prioritizing
- Synchronization
  - Controlling order of execution, visibility, atomicity
  - Locks: Can prevent data races, but watch out for deadlock!
  - Condition variables: supports communication between threads

- Most languages have similar APIs, details differ

Java: Creating threads

- Thread.create(Runnable r)
  - Or subclass the Thread class
  - Java makes it hard to create threads that access local variables (since it does not have closures)
- In practice, there are better ways
  - Use thread pools, to separate the idea of creating a thread from creating a (Runnable) task
    - May have N threads execute M > N jobs
- We'll stick with the simple idea here

Thread creation example

```
public class MyT implements Runnable {
    public void run( ) {
        …        // particular work for this thread
    }
}

Thread t = new Thread(new MyT( ));  // create thread
t.start();          // begin running thread
…                   // thread executing in parallel
t.join();           // waits for thread to exit
```

Java locks (1.4)

- Objects each have an associated lock
- Use synchronized keyword to acquire lock
  - Code blocks – synchronized (o) { … } // lock for Object o
  - Methods – synchronized foo( ) { … } // lock for this
- Thread blocks when lock held
  - Thread returns when lock is finally acquired
  - May deadlock if threads try to acquire each other's lock
- Locks sometimes referred to as mutexes

Why locks?

- #1 concern: prevent data races
- Patterns of use:
  - Enforce atomicity of shared data
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse use of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock
- Synchronization also used to ensure ordering, and visibility
  - The last is due to memory models in modern arches

Synchronization example (Java 1.4)

```
public class Example extends Thread {
  private static int cnt = 0;
  static Object value = new Object();
  public void run() {
    synchronized (value) {
      int y = cnt;
      cnt = y + 1;
    }
  }
  ...
}
```

Value, *any Java object*

***Acquires** the lock associated w/ value; only succeeds if not held by another thread, otherwise blocks*

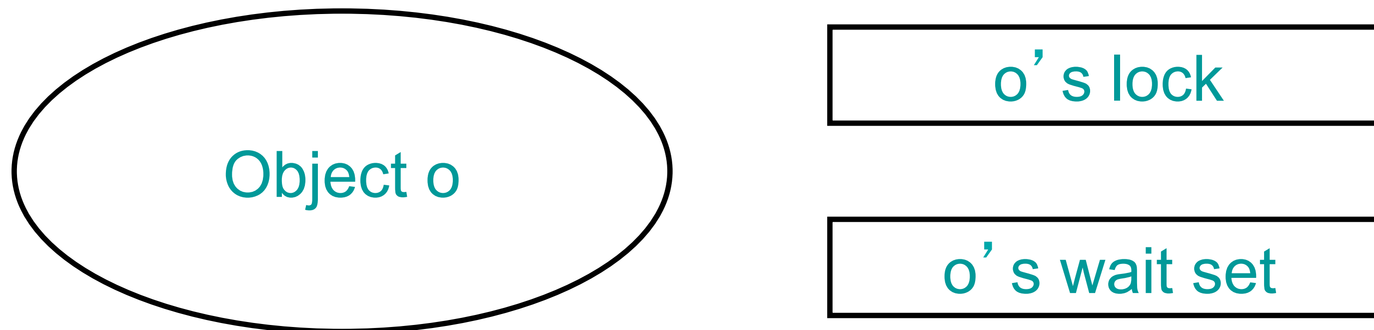
***Releases** the lock*

Condition variables

- Each condition variable represents those threads waiting for a condition to become true
  - Implemented, at least conceptually, as a wait set associated with the condition variable
- Since different threads access the variable at once, we must protect the wait set contents with a lock

Condition variables in Java 1.4

- Use synchronize on object to get associated lock



- Objects also have an associated condition variable (and thus a wait set)

Wait and NotifyAll (cont.)

- o.wait()
  - Must hold lock associated with o
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread
- o.notifyAll()
  - Must hold lock associated with o
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - This is part of the function; you don't need to do it explicitly

Using Conditions Correctly

- wait( ) **must** be called in a while loop
  - Conditions may not be met when await returns
  - Some other thread may have awoken first
    - And changed condition (e.g., consumed item in buffer)
- Avoid holding other locks when waiting
  - wait( ) only gives up lock on object you are waiting on
  - Reduces possibility of deadlock

Classic example: producer/consumer

- Want to communicate via a shared variable
  - E.g., some kind of a buffer holding messages

- One thread *produces* input to the buffer
- One thread *consumes* data from the buffer
- How do we implement this?
  - Use condition variables

Producer/Consumer in Java 1.4

```
public class ProducerConsumer {
  private boolean valueReady = false;
  private Object value;

  synchronized void produce(Object o) {
    while (valueReady) wait();
    value = o; valueReady = true;
    notifyAll();
  }

  synchronized Object consume() {
    while (!valueReady) wait();
    valueReady = false;
    Object o = value;
    notifyAll();
    return o;
  }
}
```

Java 1.5 Locks

```
interface Lock {
  void lock();
  void unlock();
  ... /* Some more stuff, also */
}
class ReentrantLock implements Lock { ... }
```

- Explicit Lock objects
  - Same as implicit lock used by synchronized keyword
- Only one thread can hold a lock at once
  - lock( ) causes thread to block (become suspended) until lock can be acquired
  - unlock( ) allows lock to be acquired by different thread

Synchronization Example (Java 1.5)

```
public class Example extends Thread {
  private static int cnt = 0;
  static Lock lock = new ReentrantLock();
  public void run() {
    lock.lock();
    int y = cnt;
    cnt = y + 1;
    lock.unlock();
    }
  }
  ...
}
```

***Lock**, for protecting the shared state*

***Acquires** the lock; only succeeds if not held by another thread, otherwise blocks*

***Releases** the lock*

ReentrantLock Class (Java 1.5)

```
class ReentrantLock implements Lock { ... }
```

- Reentrant lock
  - Can be reacquired by same thread by invoking lock( )
    - Up to 2147483648 times
  - To release lock, must invoke unlock( )
    - The same number of times lock( ) was invoked
- Reentrancy is useful
  - Each method can acquire/release locks as necessary
    - No need to worry about whether callers already have locks
  - Discourages complicated coding practices
    - To determine whether lock has already been acquired

Reentrant Lock Example

```
static int count = 0;
static Lock l =
    new ReentrantLock();

void inc() {
  l.lock();
  count++;
  l.unlock();
}
```

```
void returnAndInc() {
  int temp;

  l.lock();
  temp = count;
  inc();
  l.unlock();
}
```

- Example
  - returnAndInc( ) can acquire lock and invoke inc( )
  - inc( ) can acquire lock without having to worry about whether thread already has lock

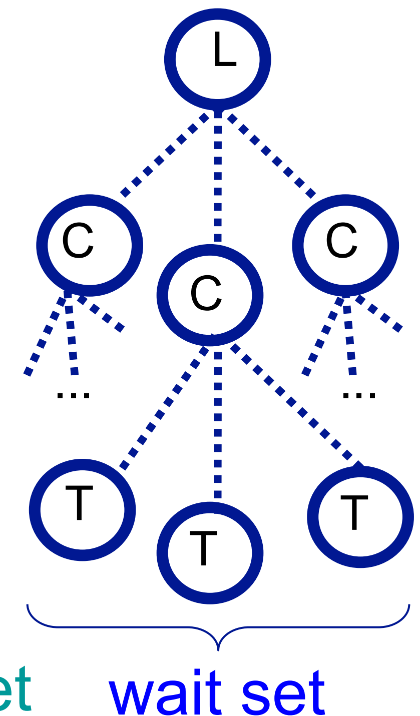
Condition Interface (Java 1.5)

```
interface Lock { Condition newCondition(); ... }
interface Condition {
  void await();
  void signalAll(); ... }
```

- Explicit condition variable objects
  - Condition variable C is created from a Lock object L by calling L.newCondition( )
  - Condition variable C is then associated with L
- Multiple condition objects per lock
  - Allows different wait sets to be created for lock
  - Can wake up different threads depending on condition

Condition – await() and signalAll()

- Calling await( ) w/ lock held
  - Releases the lock
    - But not any other locks held by this thread
  - Adds this thread to wait set for condition
  - Blocks the thread



- Calling signalAll( ) w/ lock held
  - Resumes all threads in condition's wait set
  - Threads must reacquire lock
    - Before continuing (returning from await)
    - Enforced automatically; you don't have to do it

Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
    lock.lock();
    while (valueReady)
      ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
  }
```

```
Object consume() {
    lock.lock();
    while (!valueReady)
      ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
  }
```

Use This Design

- This is the right solution to the problem
  - Tempting to try to just use locks directly
  - Very hard to get right
  - Problems with other approaches often very subtle
    - E.g., double-checked locking is broken

Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
    lock.lock();
    while (valueReady);
    value = o;
    valueReady = true;
    lock.unlock();
}
```

```
Object consume() {
    lock.lock();
    while (!valueReady);
    Object o = value;
    valueReady = false;
    lock.unlock();
}
```

Threads wait with lock held – no way to make progress

Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
    while (valueReady);
    lock.lock();
    value = o;
    valueReady = true;
    lock.unlock();
  }
```

```
Object consume() {
    while (!valueReady);
    lock.lock();
    Object o = value;
    valueReady = false;
    lock.unlock();
  }
```

valueReady accessed without a lock held – race condition

Broken Producer/Consumer Example

```
Lock lock = new ReentrantLock();
Condition ready = lock.newCondition();
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
    lock.lock();
    if (valueReady)
      ready.await();
    value = o;
    valueReady = true;
    ready.signalAll();
    lock.unlock();
  }
```

```
Object consume() {
    lock.lock();
    if (!valueReady)
      ready.await();
    Object o = value;
    valueReady = false;
    ready.signalAll();
    lock.unlock();
  }
```

what if there are multiple producers or consumers?

More on the Condition Interface

```
 interface Condition {
  void await();
  boolean await (long time, TimeUnit unit);
  void signal();
  void signalAll();
... }
```

- await(t, u) waits for time t and then gives up
  - Result indicates whether woken by signal or timeout
- signal() wakes up only *one* waiting thread
  - Same idea as notify() in Java 1.4
  - Tricky to use correctly
    - Have all waiters be equal, handle exceptions correctly
  - Highly recommended to just use signalAll()

Ruby Threads – Thread Creation

- Create thread using Thread.new
  - New method takes code block argument

    ```
    t = Thread.new { …body of thread… }
    t = Thread.new (arg) { | arg | …body of thread… }
    ```

  - Join method waits for thread to complete

    ```
    t.join
    ```

- Example

```
myThread = Thread.new {
    sleep 1                    # sleep for 1 second
    puts "New thread awake!"
    $stdout.flush              # flush makes sure output is seen
}
```

Ruby Threads – Locks

- Monitor, Mutex
  - Object intended to be used by multiple threads
  - Methods are executed with mutual exclusion
    - As if all methods are synchronized
  - Monitor is reentrant, Mutex is not
- Create lock using Monitor.new
  - Synchronize method takes code block argument

```
require 'monitor.rb'
myLock = Monitor.new
myLock.synchronize {
        # myLock held during this code block
}
```

Ruby Threads – Condition

- Condition derived from Monitor
  - Create condition from lock using new_cond
  - Sleep while waiting using wait_while, wait_until
  - Wake up waiting threads using broadcast
- Example

```
myLock = Monitor.new                    # new lock
myCondition = myLock.new_cond           # new condition
myLock.synchronize {
  myCondition.wait_while { y > 0 }      # wait as long as y > 0
  myCondition.wait_until { x != 0 }     # wait as long as x == 0
}
myLock.synchronize {
  myCondition.broadcast                 # wake up all waiting threads
}
```

Ruby Threads – Difference from Java

- Ruby thread can access all variables in scope when thread is created, including local variables
  - Java threads can only access object fields, or final local variables
- Exiting
  - All threads exit when main Ruby thread exits
  - Java continues until all non-daemon threads exit
- When thread throws exception
  - Ruby only aborts current thread (by default)
  - Ruby can also abort all threads (better for debugging)
    - Set Thread.abort_on_exception = true

Ocaml Threads – Thread Creation

- Create thread using Thread.create
  - method takes closure as its argument

    let t = Thread.create (fun x -> …body…) arg;;
  - Join method waits for thread to complete

    Thread.join t
- Example

```
let myThread = Thread.create (fun _ ->
   Unix.sleep 1;                  (* sleep for 1 second *)
   print_string "New thread awake!";
   flush Pervasives.stdout  (* flush ensures output is seen *)
);;
```

Ocaml Threads – Locks

- Mutex
  - Not reentrant
- Has lock, unlock functions

```
let myLock = Mutex.create ();;
Mutex.lock myLock;
        (* myLock held here *)
Mutex.unlock myLock
```

Ocaml Threads – Condition

- Condition module
  - Create condition directly
  - Sleep while waiting using wait (takes mutex arg)
  - Wake up waiting threads using broadcast
- Exercise: translate the Ruby code a few slides up into Ocaml code that does the equivalent thing

Compiling Ocaml threads

- Use –thread flag when compiling each .ml file
- Use threads.cma when linking
  - May also need to use unix.cma

```
ocamlc –thread unix.cma threads.cma foo.ml
```

- Ocaml threads are not parallel
  - Multiplexed on a single processor
  - Reason: concurrent GC is hard!

Alternatives to threads

- MPI – expressive, portable, but
  - Hard to partition data and get good performance
    - Temptation is to hardcode data locations, number of processors
  - Hard to write the program correctly
    - Little relation to the sequential algorithm
- OpenMP, HPF – parallelizes certain code patterns (e.g., loops), but
  - Limited to built-in types (e.g., arrays)
  - Code patterns, scheduling policies brittle

MapReduce: Distributed parallelism

- Pattern inspired by Lisp, ML, etc.
  - Many problems can be phrased this way

- Results in clean code
  - Easy to program / debug / maintain
    - Simple programming model
    - Nice retry / failure semantics
  - Efficient and portable
    - Easy to distribute across nodes

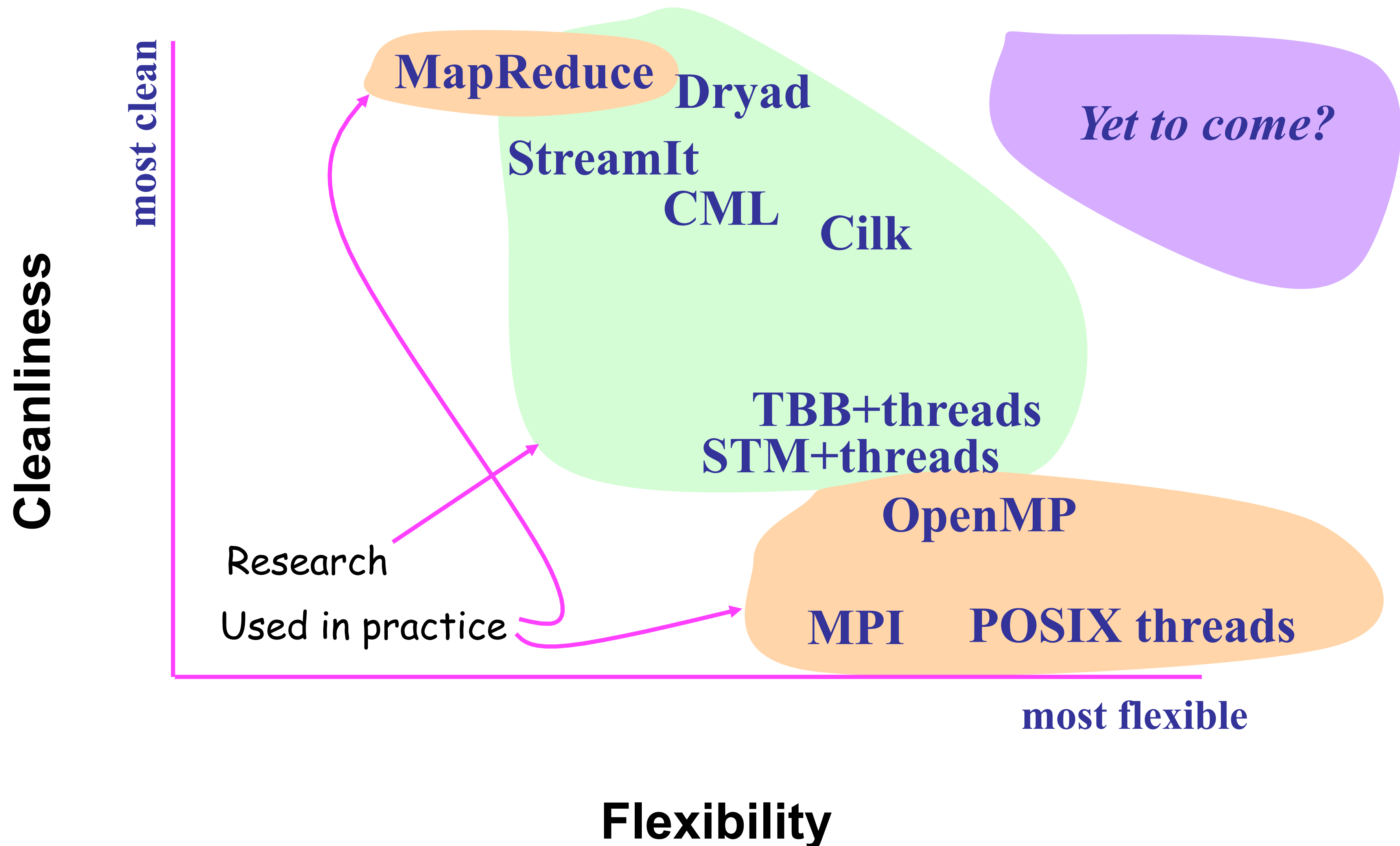
Map & Reduce in Lisp / Scheme

- (map ***f list***)
- (map square ‘(1 2 3 4))
  - (1 4 9 16)
- (reduce + ‘(1 4 9 16) 0)
  - (+ 1 (+ 4 (+ 9 (+ 16 0))))
  - 30
- (reduce + (map square ‘(1 2 3 4)) 0)

Unary operator

Binary operator

Space of Solutions



MapReduce a la Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs
- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

Count Words in Documents

- Input consists of (url, contents) pairs
- map(key=url, val=contents):
  - For each word *w* in contents, emit (w, "1")
- reduce(key=word, values=uniq_counts):
  - Sum all "1"s in values list
  - Emit result "(word, sum)"

Count, Illustrated

map(key=url, val=contents):

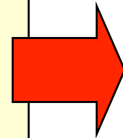
For each word *w* in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts):

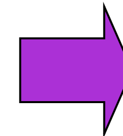
Sum all “1”s in values list

Emit result “(word, sum)”

see bob throw
see spot run

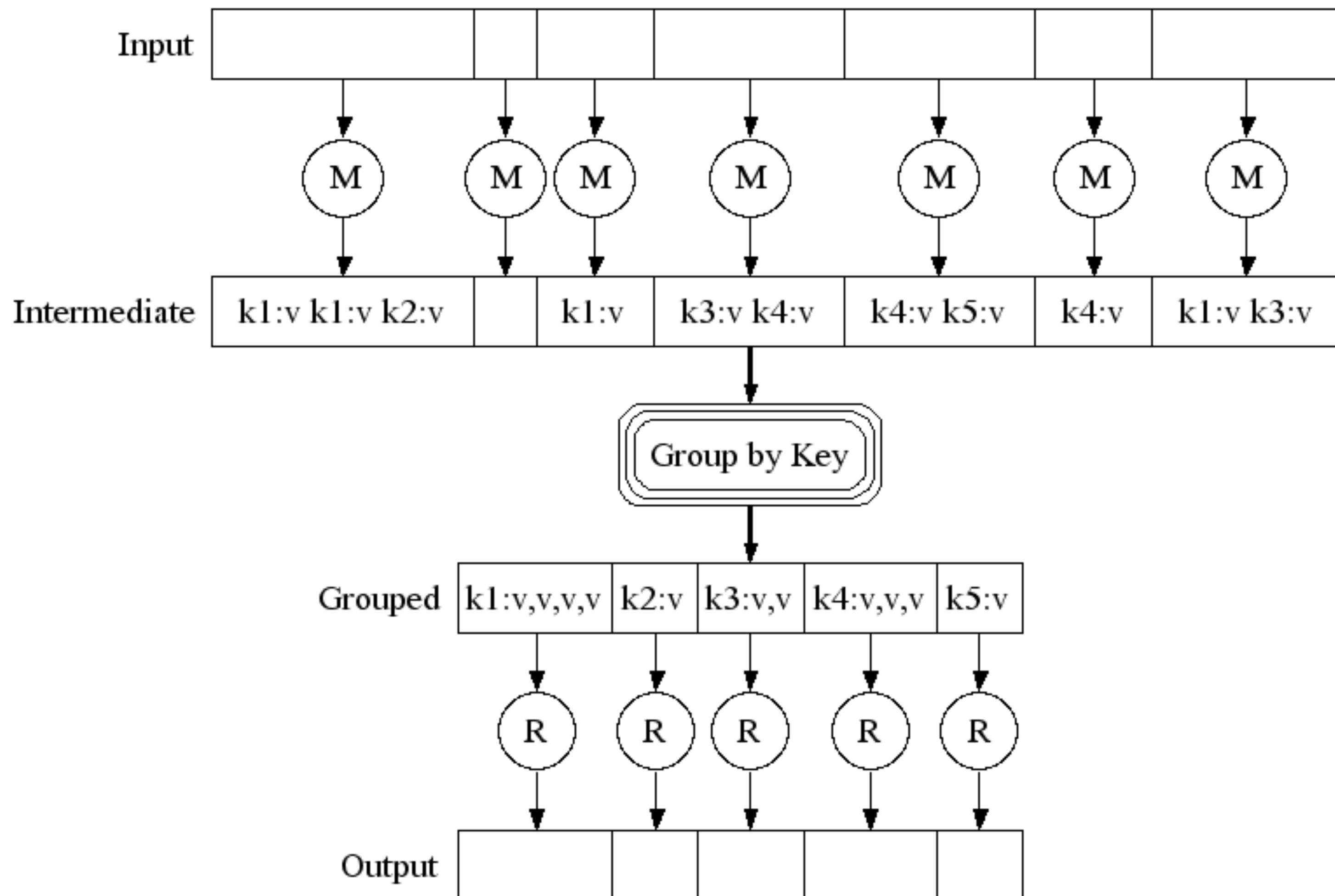


see	1
bob	1
run	1
see	1
spot	1
throw	1

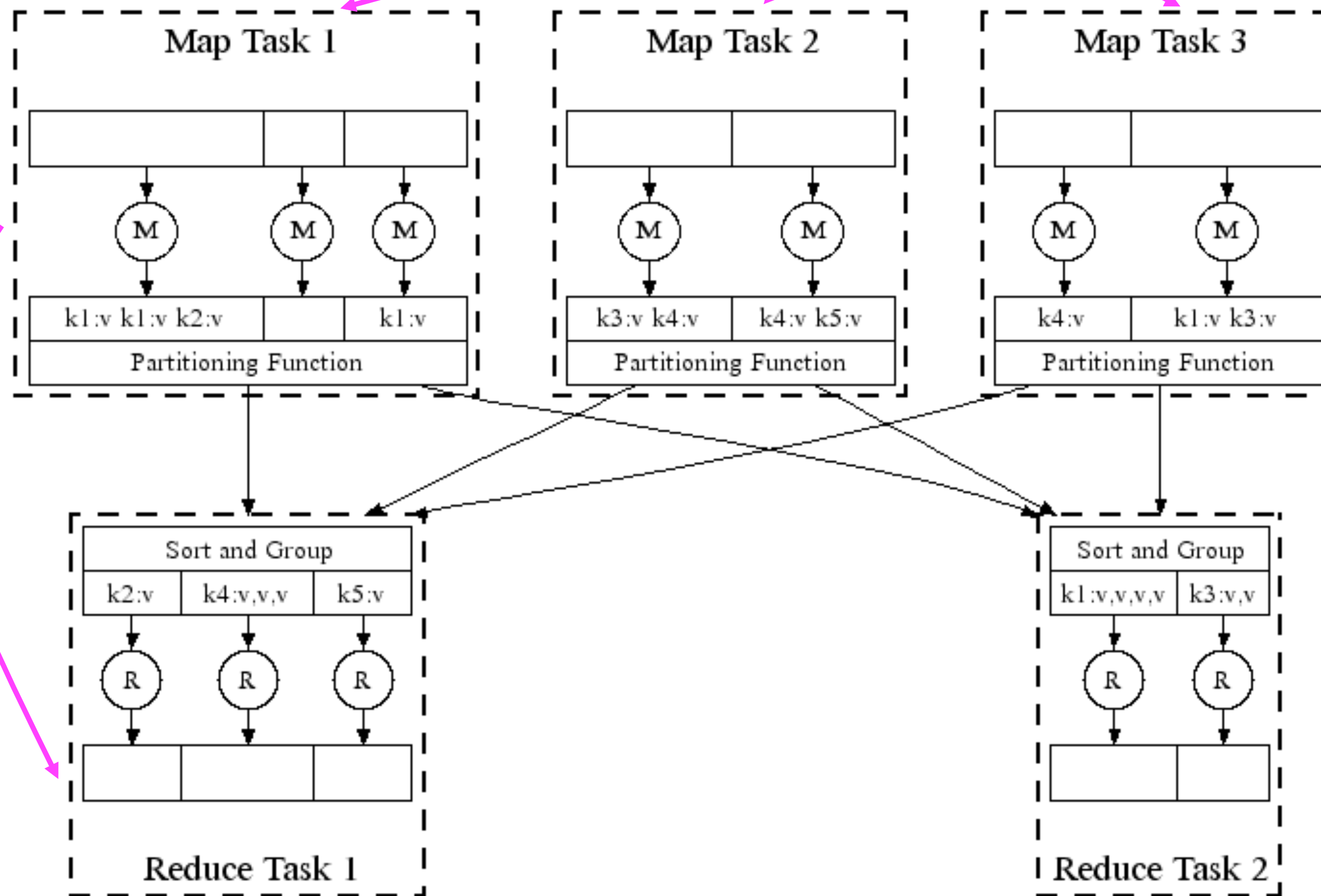


bob	1
run	1
see	2
spot	1
throw	1

Execution



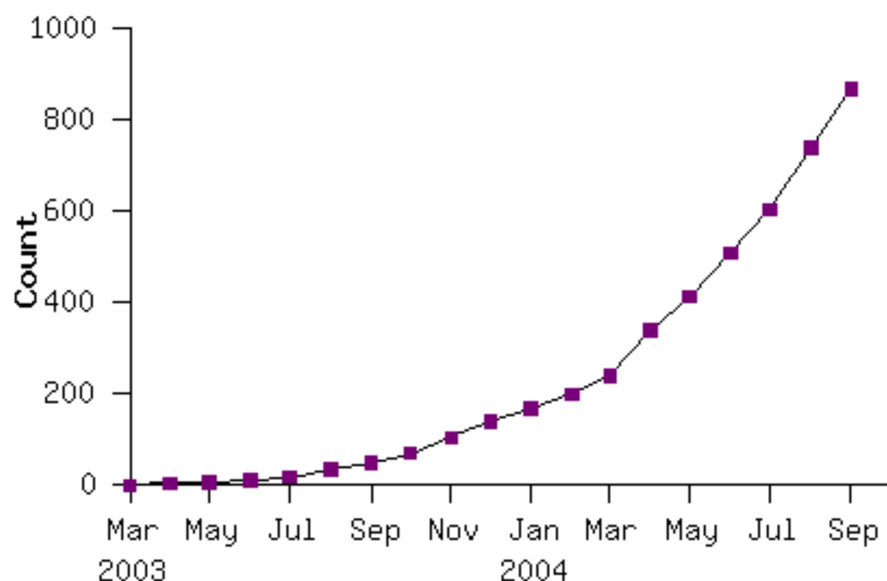
Parallel Execution



Key: no implicit dependencies between map or reduce tasks

Model is Widely Applicable

MapReduce Programs In Google Source Tree 2004



Example uses:

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

The Programming Model Is Key

- Simple control makes dependencies evident
  - Can automate scheduling of tasks and optimization
    - Map, reduce for different keys, embarassingly parallel
    - Pipeline between mappers, reducers evident
- map and reduce are pure functions
  - Can rerun them to get the same answer
    - In the case of failure, or
    - To use idle resources toward faster completion
  - No worry about data races, deadlocks, etc. since there is no shared state

Compare to Dedicated Supercomputers

- According to Wikipedia, in 2009 Google uses
  - 450,000 servers (2006), mostly commodity Intel boxes
  - 2TB drive per server, at least
  - 16GB memory per machine
  - More recent details are kept secret by Google

- More computing power than even the most powerful supercomputer